Computers Math. Applic. Vol. 25, No. 7, pp. 3–13, 1993
Printed in Great Britain.

0097-4943/93 $6.00 + 0.00

FAST, PORTABLE, AND RELIABLE ALGORITHM FOR THE CALCULATION OF HALTON NUMBERS*

MIROSLAV KOLÁŘ AND SEAMUS F. O'SHEA
Department of Chemistry, University of Lethbridge
Lethbridge, Alberta, Canada T1K 3M4

(Received October 1992)

Abstract—We give a recursive algorithm closely based on the definition of Halton numbers—reflection in the radical point of the digits of an integer in an arbitrary-base positional notation—which is, unlike the Halton's short algorithm, not at all affected by the round-off error, and much faster than the recent improvement of the Halton's algorithm by Berblinger and Schlier. Some applications of the Halton numbers are discussed.

1. INTRODUCTION

Sequences of Halton vectors are one of several types of quasi-random sequences designed for efficient Monte Carlo calculations of multi-dimensional integrals. In this note, we present an algorithm for the generation of the components of Halton vectors that is much faster than the presently used generators and, at the same time, fully portable and resistant to round-off error.

The n^{th} term of an s-dimensional Halton sequence is a vector whose components are Halton numbers, $H_n(b_k)$, corresponding to s different mutually prime bases b_k, $k = 1, \ldots, s$. Halton numbers [1] are the generalization of the van der Corput numbers [2] introduced originally for $b_k = 2$ only. Their practical value for the Monte Carlo calculation of multi-dimensional volumes has recently been verified by Berblinger and Schlier [3].

Any integer n can be written in the base-b notation as

$$n = d_j \cdots d_2\, d_1 d_0\,.\,0 \text{ (base } b) = d_0 + d_1 b + d_2 b^2 + \cdots + d_j b^j. \tag{1}$$

The n^{th} Halton number $0 \le H_n(b) < 1$ is then defined as

$$H_n(b) = 0\,.\,d_0\, d_1 d_2 \cdots d_j \text{ (base } b) = \frac{d_0}{b} + \frac{d_1}{b^2} + \frac{d_2}{b^3} + \cdots + \frac{d_j}{b^{j+1}}. \tag{2}$$

It would be wasteful of CPU time to perform this reflection in radix point repeatedly for each n. To go from $H_n(b)$ to $H_{n+1}(b)$ simply means the addition to $H_n(b)$ of $1/b = 0.1$ (base b), with a rightward "carry," i.e., a carry in the opposite direction than that in normal addition. Halton [1] gave for this recursion the following short algorithm:

$$\begin{aligned} &y := 1/b \\ &x := 1 - H_n \\ &\textbf{while } \; x \le y \; \textbf{ do } \; y := y/b \\ &H_{n+1} := (b+1)\,y - x \end{aligned} \tag{3}$$

with initialization $H_0 = 0$. In floating point arithmetic, round-off error can result in the $x \le y$ condition in Algorithm (3) not being evaluated properly. That can lead to near-repetitions of Halton numbers, or in the worst case to infinite loops for some bases, as discussed in more detail

*This research was supported in part by a grant from the Natural Sciences and Engineering Research Council of Canada.

in [3]. Halton and Smith [4] solved this problem by introducing a small error tolerance E and replacing the above condition with $x < y + E$. However, to make a suitable choice of E is a rather complicated process as E is dependent on the set of bases $\{b_k\}$ and also on the precision used, which results in a nonportable generator. For example, Fox [5] needed a full four pages to describe the choice of a suitable E.

A better solution is that of Berblinger and Schlier [3], who represented all the rational numbers occurring in Algorithm (3) as ratios of integers. Using a common denominator, one can write

$$H_n = N_n/D_n, \quad x = X/D_n, \quad y = Y/D_n.$$

Then Algorithm (3) can be rewritten as

$$\begin{aligned}
&X := D_n - N_n \\
&\textbf{if } X = 1 \textbf{ then } \{\, N_{n+1} := 1;\ D_{n+1} := b\,D_n;\ \textbf{return} \,\} \\
&Y := D_n/b \\
&\textbf{while } X \le Y \textbf{ do } Y := Y/b \\
&N_{n+1} := (b+1)Y - X \\
&D_{n+1} := D_n
\end{aligned} \tag{4}$$

with initialization $N_0 = 0$, $D_0 = 1$. This algorithm is fully portable and should produce sequences independent of the computer used. The corresponding C code can be found in Appendix A. It is equivalent to the FORTRAN code of [3]. We will use this program as a standard with which to compare the speed of our faster algorithm presented below. Although N_n and D_n always assume integer values, it is convenient to declare them as `double` on machines where floating point arithmetic is faster than integer arithmetic, and it also allows an increase in the maximum length of the generated sequence beyond $2^{31}/b$ (assuming that an `int` variable has 32 bits).

Note that the condition $X = 1$ in Equation (4) is satisfied only if $d_0 = d_1 = \cdots = d_j = b - 1$ in Equation (1). The condition $X \le Y$ is satisfied when a carry is required in all other cases (when only some digits are equal to $b-1$), and each rightward carry then corresponds to a single execution of the body of **while**. For larger bases b, no carry is necessary for most values of n. In this most common case, two decision-making evaluations, one addition, two subtractions, one division and one multiplication are made.

2. FAST NEW ALGORITHM

To increase the speed of a Halton number generator, one has to try to minimize the number of operations required for the generation of the next Halton number under any circumstances (no carry, or carry by an arbitrary number of places). We have achieved this by following closely the original definition (2) of the Halton numbers. Let us introduce the following array of "remainders" r_i:

$$\begin{aligned}
&r_j = 0 \\
&r_{i-1} = (d_i + r_i)/b; \qquad i = j, j-1, \ldots, 1.
\end{aligned} \tag{5}$$

Then the Halton number of Equation (2) can be written as

$$H_n = (d_0 + r_0)/b, \tag{6}$$

and Equations (5) and (6) lead to the following algorithm:

$$\begin{aligned}
&\textbf{Assume that } d_i \textbf{ and } r_i \textbf{ have values corresponding to } H_n \\
&l := 0 \\
&\textbf{while } d_l = b - 1 \textbf{ do } \{\, d_l := 0;\ l := l + 1 \,\} \\
&d_l := d_l + 1 \\
&\textbf{if } l \ge 1 \textbf{ then } r_{l-1} := (d_l + r_l)/b \\
&\textbf{if } l \ge 2 \textbf{ then for } i = l-1 \textbf{ step } 1 \textbf{ until } 1 \textbf{ do } r_{i-1} := r_i/b \\
&H_{n+1} := (d_0 + r_0)/b
\end{aligned} \tag{7}$$

with initialization $d_i = r_i = 0;\ i = 0, 1, \ldots, D$. This will allow for the generation of at most b^D Halton numbers.

Table 1. Comparison of CPU time in seconds needed for the calculation of the first 1 million Halton numbers by the functions listed in Appendices A, B, and C. Timing was done on DECstation 5000/125.

base	haltonBS()	halton()	halton2()
2	3.94	2.62	1.15
3	3.45	1.89	
5	3.20	1.51	
7	3.12	1.38	
11	3.05	1.28	
13	3.05	1.26	
17	3.03	1.23	
19	3.02	1.22	
23	3.00	1.21	
107	2.98	1.15	
659	2.97	1.14	

The price paid for faster performance is somewhat larger memory space required. Instead of two variables for the storage of D_n and N_n, we now need two arrays r_i and d_i of $D+1$ elements. However, this is no serious problem, as with $D = 20$ and $b = 3$, one can already generate almost 3.5 billion Halton numbers. In the C programming language, Algorithm (7) can easily be coded in such a way that, in the most frequent no-carry situation, only one decision-making condition has to be evaluated, and one addition and one division performed, as can be seen in Appendix B. Therefore, one can expect a speedup by a factor of more than 2 for larger bases b. This is confirmed by the times required to calculate the first 1 million Halton numbers that are presented in Table 1, where `haltonBS()` is the function of Appendix A based on the Berblinger and Schlier's Algorithm (4), and `halton()` is the function of Appendix B based on our Algorithm (7). Algorithm (7) cannot be "derailed" by the round-off error, because digits d_i are exact at all times, and remainders r_i are recalculated every time when the respective digits do change. We have checked that the difference between the numbers produced by `halton()` and those produced by `haltonBS()` was within the machine precision of the computer used.

When $b = 2$, the digit array r_i is utilized especially inefficiently as every digit d_i can assume only the values 0 and 1. In this case, the role of d_i can be easily played by individual bits of an integer variable. One can then rewrite Equation (2) as

$$H_n(b) = \tilde{n}/2^D;\quad \tilde{n} = d_0 2^{D-1} + d_1 2^{D-2} + \cdots + d_j 2^{D-j-1}. \tag{8}$$

In binary notation, $\tilde{n}$ is an integer whose last D bits contain last D bits of n in *reversed* order. Thus, the array r_i need not be maintained in this case, as the next Halton number H_{n+1} can be produced by adding 2^{D-1} to $\tilde{n}$ with a rightward carry if necessary, followed by a single division of the result by 2^D. The C code for Algorithm (7) utilizing Equation (8) is presented in Appendix C. It uses fast bit manipulations available in C. $\tilde{n}$ is stored in variable `inv_n`, and 2^D = `MAXHLT1`. Also this code is highly portable: there is a single parameter, `BITSPERBYTE`, to be checked and possibly modified when porting the program to another machine. As can be seen from Table 1, this code, `halton2()`, is more than twice faster than the generic function `halton()` of Appendix B for $b = 2$.

Note that we have actually not utilized the maximum available number of bits of `inv_n`. One could still increase the values of `HI_BIT` and `MAXHLT1` by a factor of 2, and start with `bit=1` in the `for` loop of `init_halton2()`. This would increase the maximum number of Halton numbers generable by this function to more than 4 billion. However, we have noticed that employing the sign bit in this way slows down the calculation somewhat: from the present 1.15 CPU sec to 1.23 CPU sec needed for 1 million Halton numbers.

Algorithm (7) also allows for an easy implementation of the scrambled Halton numbers [6]. One needs only change the definition of the remainders r_i in Equation (5) to

$$r_{i-1} = [\pi(d_i) + r_i]/b,$$

where π is a suitable permutation on the digits d_i [6]. That would add to Algorithm (7) a single search in a fixed-valued array representing π.

To generate a full Halton vector with components corresponding to different bases b_k, there are, in principle, two choices. One possibility is to introduce two-dimensional arrays d_{ki} and r_{ki} and change k periodically on each call to `halton()`. This slows down the computation. For example, it takes 2.6 sec to calculate 25000 40-dimensional Halton vectors (i.e., again 1 million Halton numbers) using the lowest 40 prime bases. For smaller number of dimensions s, it would be faster to have separate functions for each b_k and for example in C to switch between them through an array of pointers to functions. This would also allow for an easy incorporation of the fast code of `halton2()` for $b_k = 2$.

Even such a multiple-basis Halton number generator is still faster than some good common pseudo-random number generators. For example, in order to generate 1 million random numbers, Marsaglia's portable generator RANMAR [7] needs 3.63 sec and DEC's best standard (non-portable) generator DRAND48/ERAND48, 6.28/10.04 sec. Only the rather poor standard UNIX generators RANDOM and RAND are faster, requiring 1.33 and 1.02 sec, respectively (these times already include a division by `(MAXINT+1)` to get random numbers between 0 and 1).

3. APPLICATIONS

On the basis of the calculation of the test integral (cf. [8, p. 406])

$$I = \int_0^1 \cdots \int_0^1 \prod_{k=1}^{s} |4x_k - 2|\, dx_1 \cdots dx_s = 1, \tag{9}$$

Fox [5] concluded that Halton sequences are not especially suitable for dimensions larger than $s \approx 10$. Comparing the results such as those for $s = 40$ of the second and third rows of Table 2, he found the Faure numbers [9] to be far superior to Halton numbers for larger dimensions. The third row of Table 2 was obtained with the Halton integration grid with all Halton vector components starting from the beginning ($n0 = 1$ in `init_halton()` of Appendix B). Neither Fox, nor Davis and Rabinowitz [8] (for $s = 25$) noted that the poor performance of the Halton numbers for Integral (9) is caused just by the first few members of the Halton sequence that are too close to the corners of the unit hypercube over which the integration is done. Let us denote by f_n the values of the integrand of Equation (9) obtained for $s = 40$ with the n^{th} Halton vector, i.e., $x_k = H_n(b_k)$, $k = 1, \ldots, 40$. Here b_k are the lowest 40 primes $2, 3, 5, \ldots, 173$. Then $f_1 = 0$, $f_2 = 6.07 \cdot 10^8$, $f_3 = 6.85 \cdot 10^7$, $f_4 = 4.40 \cdot 10^6$, $f_5 = 1.36 \cdot 10^6$, $f_6 = 6.94 \cdot 10^4$, $\ldots$, $f_{10} = 113.4$, $\ldots$, $f_{20} = 0.007806$, etc. The Monte-Carlo estimate of Integral (9) using N grid points is

$$I_{\text{est}} = \frac{1}{N} \sum_{i=1}^{N} f_i.$$

Because the integrand is always positive, $I_{\text{est}} > f_2/N$. Thus, $N \gtrsim 6 \cdot 10^8$ would be needed to get the Halton estimate to the vicinity of the true value $I = 1$. The fourth row of Table 2 shows that for the range of N used, I_{est} is essentially determined by the first five members of the Halton sequence.

When integrating (9), one can get huge errors in any integration scheme, except perhaps for small s, because the variance σ^2 of Integral (9) increases rapidly with s. Analytical calculation gives

$$\sigma^2 = \int_0^1 \cdots \int_0^1 \prod_{k=1}^{s} (4x_k - 2)^2\, dx_1 \cdots dx_s - I^2 = \left(\frac{4}{3}\right)^2 - 1.$$

Table 2. Comparison of the estimates I_{est} for the Integral (9) for $s = 40$ obtained with N 40-dimensional grid points generated by various generators.

N	500	1000	7000	$2 \cdot 10^4$	$4 \cdot 10^4$	10^5	$2 \cdot 10^5$	10^6
FAURE[a)]	0.193	0.208	0.330	0.637	0.682	0.675	...	...
HALTON ($n0 = 1$)	1,362,765	681,382	97,341	34,069	17,035	6815	3408	682.2
$\frac{1}{N}\sum_{i=2}^{5} f_i$	1,362,587	681,294	97,328	34,065	17,032	6813	3406	681.3
$n0 = 10$	0.677	0.392	0.423	0.423	0.527	0.934	0.883	0.786
$n0 = 5 \cdot 10^5$	0.005	0.547	0.512	0.427	0.390	0.425	0.509	0.614
random $n0$	0.846 ±4.11	0.832 ±3.27	0.997 ±2.14	0.957 ±1.22	1.030 ±1.23	1.084 ±1.69	1.013 ±0.91	0.993 ±0.34
	0.855 ±5.34	0.793 ±3.31	1.132 ±7.77	1.142 ±4.39	...	...	...	...
RANMAR	0.725 ±2.16	0.984 ±4.19	0.992 ±1.58	0.946 ±0.90	0.960 ±0.71	0.956 ±0.47	0.952 ±0.40	0.986 ±0.24
	0.677 ±2.02	0.871 ±3.21	1.002 ±2.19	1.036 ±1.73	...	...	...	...

[a)] From Table 1 of [5].

For $s = 40$, $\sigma = 315.335$. The Central Limit Theorem (e.g., [8, p. 387]) states that

$$\text{probability}\left(|I_{est} - I| \leq \frac{\lambda\sigma}{\sqrt{N}}\right) = \text{erf}\left(\frac{\lambda}{\sqrt{2}}\right) \tag{10}$$

when the integration grid of N points is selected at random. However, even with the above large value of σ, the huge values for I_{est} obtained with the Halton sequence with $n0 = 1$ are rather exceptional. For example, from Equation (10) we have that the probability that $I_{est} > 681$ for Integral (9) with $s = 40$ and $N = 10^6$ (cf. $I_{est} = 682.2$ for Halton grid with $n0 = 1$) is equal to $1 - \text{erf}\left(\frac{680000}{\sigma\sqrt{2}}\right) < 1.62 \cdot 10^{-1009784}$. In principle, there are grids that could give even worse estimates of the Integral (9) than the Halton grid does. If one point of a grid would come arbitrarily close to a corner of the integration volume hypercube, than $I_{est} \gtrsim 2^{40}/N \approx 10^{12}/N$. However, this could happen with an even much more negligible probability than the one in the above for $I_{est} > 681$.

Row 5 of Table 2 shows that skipping just the first nine members of the Halton sequence gives values of I_{est} comparable to (better than) those obtained with the Faure sequence. At the same time, according to the timings of [5] one can generate the Halton numbers much faster, especially with our fast algorithm, than the Faure numbers. However, the convergence to the true result $I = 1$ still seems to be quite slow. Things are not improved by skipping a larger portion of the beginning of the Halton sequence as shown in the sixth row of Table 2.

We have found that better results can be obtained when using modified Halton sequences with a random selection of the values of $n0$ different for different components of the Halton vector, and then taking the averages over a series of estimates obtained in this way. We have been selecting $n0 \in (1, 50000)$. The averages and standard deviations of a series of 500 such calculations can be found in the 7^{th} and 8^{th} row of Table 2. One can see that the obtained standard deviations are of the order of $\sigma/\sqrt{N}$ as expected. Five sets of values of I_{est} of the total 500 are shown in Table 3. According to Equation (10), the probability that for $N = 10^6$, $0.7873 \leq I_{est} \leq 1.2127$, is 50%. The modified Halton sequence with random selection of $n0$ did better than that: 62.4% of all the 500 values of I_{est} for $N = 10^6$ that were used to obtain the last number in the 7^{th} row of Table 2 lay in the above interval. All 500 values were contained in the interval (0.56,3.74). The 9^{th} and 10^{th} row of Table 2 show how the averages and standard deviations of the preceding two rows changed when another 500 calculations were added to the previous sample.

From Equation (10), for $N = 2 \cdot 10^4$, 50% of all estimates should be in the interval $0 \leq I_{est} \leq 2.5040$. Of the 1000 values of I_{est} calculated, actually full 94.2% lay in this interval. Large

Table 3. Examples of the values of I_{est} obtained with the Halton generator with different random $n0$ (between 1 and 50000) for different Halton vector components, and with the RANMAR generator with different seeds. These values are part of the data used to calculate the averages near the bottom of Table 2.

N	500	1000	7000	$2 \cdot 10^4$	$4 \cdot 10^4$	10^5	$2 \cdot 10^5$	10^6
HALTON	0.176	0.089	6.519	5.193	2.814	2.057	1.256	0.925
with	12.62	6.313	1.243	0.752	2.052	1.246	1.197	1.097
random	0.494	0.327	0.611	0.505	0.493	0.356	0.445	0.889
$n0$	0.461	0.234	0.463	1.302	1.296	1.067	0.882	0.971
	0.028	0.088	0.132	0.586	0.507	1.156	0.986	1.543
RANMAR	18.21	9.185	3.222	1.395	1.374	0.978	0.782	0.734
	0.211	0.125	1.369	1.165	0.826	0.577	0.690	0.767
	0.172	0.247	1.741	1.131	0.933	1.283	0.950	1.389
	0.391	0.554	3.063	1.568	1.102	0.796	0.754	1.193
	0.853	0.551	0.285	0.517	0.758	0.787	0.766	0.948

standard deviations for all 1000 calculations (10^{th} row) do not yet represent converged values. For the first 850 calculations, the deviations were equal to 3.25, 2.59, 1.78, and 1.12 for $N = 700$, 1000, 7000, and $2 \cdot 10^4$, respectively. In any case, it seems that one can get quite good results by taking the average of a large number of calculations done with a smaller value of N, such as $2 \cdot 10^4$.

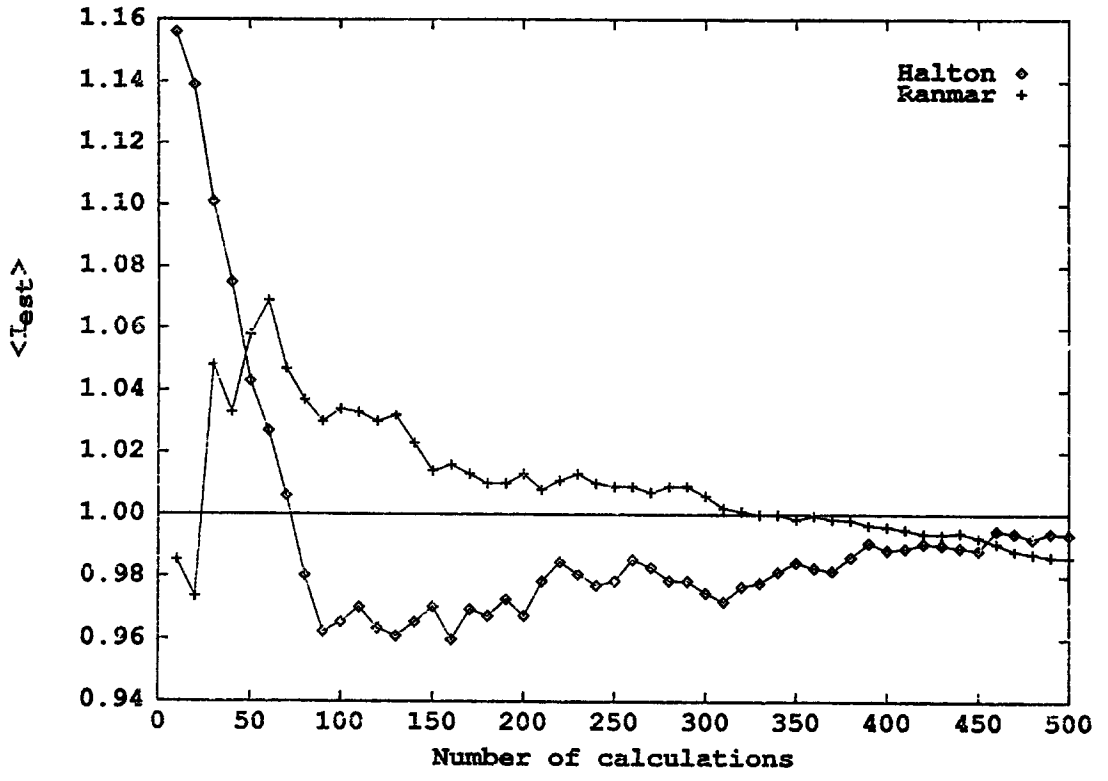
For comparison, at the bottom of Table 2 one can find estimates obtained with the pseudo-random number generator RANMAR [7] by averaging over samples of I_{est} of the same size as those for the Halton generator with randomly selected $n0$. In this case, we generated different sequences of random numbers by selecting randomly the two seeds of RANMAR. The overall scatter of individual values of I_{est} seems to be somewhat smaller than for the Halton sequence. Also, for $N = 10^6$, there is an even larger percentage of the values of I_{est} contained within the theoretical 50% probability interval (0.7873, 1.2127), the full 82%. But all 500 values were contained in the interval (0.66,3.95) that is slightly wider than the same interval for the Halton numbers. For $N = 2 \cdot 10^4$, the theoretical 50% probability interval $(0, 2.5040)$ contained 95.1% of all 1000 values.

Finally, Figure 1 shows how the cumulative averages of I_{est} do change as the results of new calculations are added. One can see that it is really difficult to calculate Integral (9) to a high precision. Even the large number of calculations that we have done was not enough to eliminate the oscillation of $< I_{est} >$ about the true value $I = 1$. This is linked to the quite large scatter of the values of I_{est} for a single N as observed in Tables 2 and 3, and all this is the manifestation of the large value of σ.

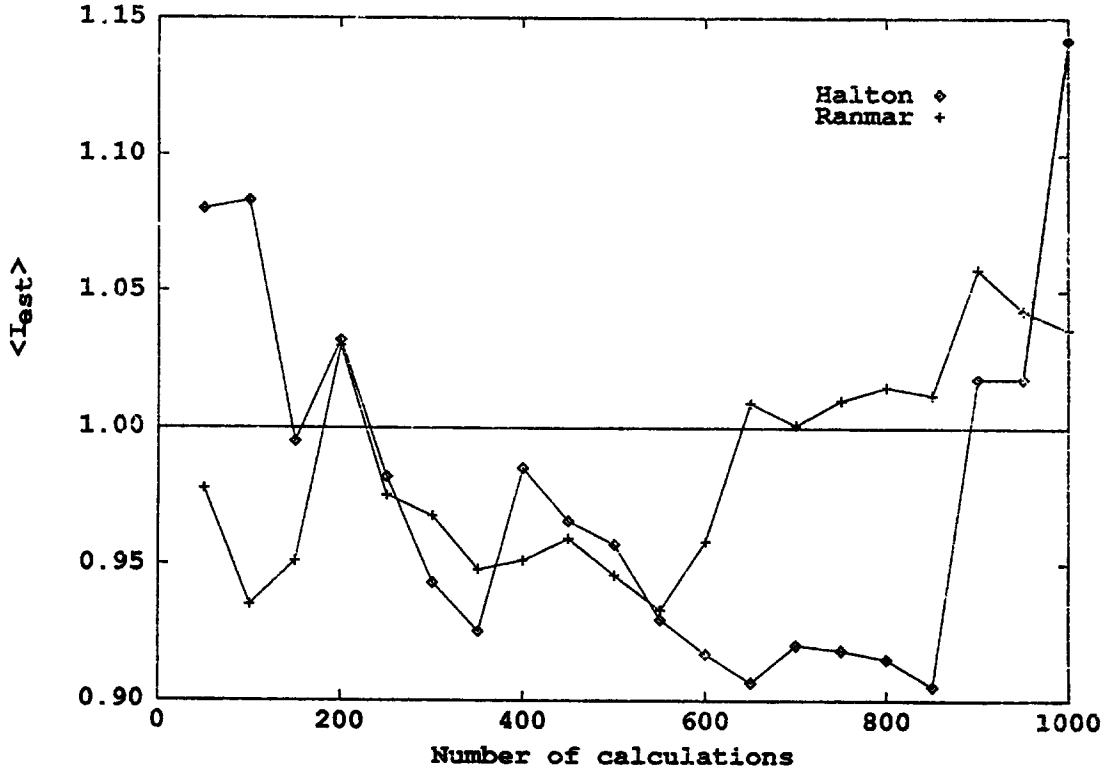
Because of this large scatter of the results for I_{est}, even after the rather extensive set of calculations we have done (comprising altogether $2.04 \cdot 10^{10}$ calls to each HALTON and RANMAR), it is difficult to make a definitive conclusion which of the two generators is better for Integral (9). One of the strong advantages of the Halton numbers is the speed of our fast algorithm. Using it, the above calculations were faster by a factor of 1.35 than when using the RANMAR generator. Our algorithm (as coded in Appendices B and C) makes it also possible to start at an arbitrary point of a Halton sequence with no additional computational overhead. This makes the use of Halton numbers with random selection of $n0$'s quite attractive. In any case, we have demonstrated that the Halton numbers that are known to be very effective in other cases [3], can be made to perform very well also in the case of the difficult Integral (9) for which they were previously believed to fail completely.

Another possible application of a fast Halton number generator is to increase the period and improve the quality (uniformness) of various pseudo-random number generators using Lemmas 1 and 2 of [10]. In short, when $W_1, \ldots, W_l$ are l independent discrete random variables (generated by l different generators) such that W_1 is uniform between 0 and d, then

$$\left(\sum_{i=1}^{l} W_i \right) \bmod d \tag{11}$$



(a) $N = 10^6$.



(b) $N = 2 \cdot 10^4$.

Figure 1. Cumulative averages of the estimates I_{est} for the Integral (9) for $s = 40$.

is also uniform between 0 and d, and the period of such a combined generator is the least common multiple of the periods of all l generators. Note that only one of the l random variables need be uniform to obtain a combined generator that is uniform.

Halton numbers are highly uniform and their theoretical period is infinite, while the actual period depends on their computer representation. We have tried to combine $\tilde{n}$ of Equation (8) with DEC's RANDOM according to Equation (11). Using such a combined generator in the quantum path-integral Monte Carlo (Metropolis-type) simulation of a harmonic oscillator leads to a big improvement of the results over those obtained with RANDOM only [11]. RANDOM itself proved to be very bad for this purpose, and the Halton sequence seems to be completely unsuitable for Metropolis sampling (this can be considered as a preliminary answer to the Berblinger and Schlier's question [3] about the usefulness of Halton numbers for Metropolis sampling). However, surprisingly, the combination of RANDOM with our algorithm of Appendix C for $b = 2$ gave results almost as good as those obtained with RANMAR.

4. SUMMARY

We have found a new algorithm for the generation of Halton numbers that, for an average selection of prime Halton bases, leads to a speedup by factor 2 compared to the previously known algorithm. It is fully portable and not affected by the round-off error. This algorithm can serve as the basis of an efficient quasi-Monte Carlo multi-dimensional integrator, and combined with some pseudo-random number generators can increase their period and improve their uniformity.

REFERENCES

1. J.H. Halton, On the efficiency of certain quasi-random sequences of points in evaluating multi-dimensional integrals, *Numerische Mathematik* **2**, 84–90 (1960).
2. J.G. van der Corput, Verteilungsfunktionen, *Proc. K. Ned. Akad. v. Wet.* **38**, 813–821 (1935).
3. M. Berblinger and C. Schlier, Monte Carlo integration with quasi-random numbers: Some experience, *Computer Physics Communications* **66**, 157–166 (1991).
4. J.H. Halton and G.B. Smith, Radical-inverse quasi-random point sequence, Algorithm 247, *Communications of the ACM* **7** (12), 701–702 (1964).
5. B.L. Fox, Implementation and relative efficiency of quasirandom sequence generators, Algorithm 647, *ACM Transactions on Mathematical Software* **12** (4), 362–376 (1986).
6. E. Braaten and G. Weller, An improved low-discrepancy sequence for multidimensional quasi-Monte Carlo integration, *Journal of Computational Physics* **33**, 249–258 (1979).
7. G. Marsaglia, Florida State University Report: FSU-SCRI-87-50, (1987).
8. P.J. Davis and P. Rabinowitz, *Methods of Numerical Integration*, Academic Press, Orlando, FL, (1984).
9. H. Faure, Discrépance de suites associées à un système de numération (en dimension s), *Acta Arithmetica* **XLI**, 337–351 (1982).
10. P. L'Ecuyer, Efficient and portable combined random number generators, *Comm. of the ACM* **31** (6), 742–750 (1988).
11. M. Kolář and S.F. O'Shea, (to be published).

APPENDIX A

C CODE FOR ALGORITHM OF EQUATION (4)

```
static int b;
static double Nn, Dn;

double init_haltonBS(base, n0) int base, n0;
/*=======================================*/
{
 /* Must be called prior to haltonBS().
    haltonBS() will then start with n0-th Halton number */

 double haltonBS();

 b = base;
 Nn = 0;
 Dn = 1;
 while(--n0 > 0) haltonBS();
}

double haltonBS()
/*==============*/
{
 /* On each call, next Halton number is returned */

 register double Y;
 register double X;

 X = Dn - Nn;
 if(X == (double)1.) { Nn = 1.;
                       Dn *= b;
                     }
 else   { Y = Dn / b;
          while(X <= Y) Y /= b;
          Nn = (1. + b) * Y - X;
        }
 return Nn / Dn;
}
```

APPENDIX B

C CODE FOR ALGORITHM OF EQUATION (7)

```
#define D  20  /* to generate at most b^D Halton numbers */

static int b;
static unsigned int d[D + 1];
static double r[D + 1];

void init_halton(base, n0) int base, n0;
/*====================================*/
{
 /* Must be called prior to halton().
    halton() will then start with n0-th Halton number */

 unsigned int last, i;

 b = base;
 if(n0 < 1) n0 = 1;
 --n0;

 /* Invert the order of digits of (n0-1) written in the
    base-b notation:  */
 last = 0;
 while(n0 >= b) { d[last++] = n0 % b; n0 /= b; }
 d[last] = n0;
 for(i = last + 1; i < D; i++) d[i] = 0;
 r[D] = 0;
 for(i = D; i > 0; i--) r[i - 1] = (d[i] + r[i]) / b;
}

double halton()
/*============*/
{
 register unsigned int l = 0;
 register unsigned int i;

 if(++d[0] == b)
    { do d[l++] = 0; while(++d[l] == b);
      r[l - 1] = (d[l] + r[l]) / b;
      for(i = l - 1; i > 0 ; i--) r[i - 1] = r[i] / b;
      return r[0] / b;
    }
 else return (d[0] + r[0]) / b;
}
```

APPENDIX C

C CODE FOR ALGORITHM OF EQUATION (7) FOR $b = 2$ USING EQUATION (8) AND BIT OPERATIONS

```
#define BITSPERBYTE     8   /* check this value for your machine */
#define BITS(type)      (BITSPERBYTE * (int)sizeof(type))
#define HIBITI  (1 << BITS(int) - 1)
#define MAXINT  (~HIBITI)
/* on some systems, the above definitions can be found in a standard
   include file such as <values.h> for UNIX/SUN or Ultrix/DECstation */

static unsigned int HI_BIT = (1 << BITS(int) - 2);
static double MAXHLT1;  /*  = (MAXINT + 1.);  see below        */
                        /*  = 2147483648 for 32 bit integers   */
    /* allows for the generation of MAXHLT1 - 1 Halton numbers */
static unsigned int inv_n;  /* to store the inverted order of bits */

void init_halton2(n0) unsigned n0;
/*==============================*/
{
 int bit;

 /* Must be called prior to halton2().
    halton2() will then start with n0-th Halton number to the base 2 */

 MAXHLT1 = MAXINT + 1.;

 if(n0 < 1) n0 = 1;
 --n0;

 /* Invert the order of bits of n0, and store the result in inv_n: */
 inv_n = n0 & 01;
 for(bit = 2; bit < BITS(int); bit++)
 { inv_n <<= 1; inv_n |= ((n0 >>= 1) & 01); }
}

double halton2()
/*=============*/
{
 register unsigned int bit;

 if(inv_n & HI_BIT)
    { /* at least one carry needed: */
      inv_n &= ~HI_BIT;
      bit = HI_BIT >> 1;
      while(inv_n & bit) { inv_n &= ~bit; bit >>= 1; }
      inv_n |= bit;
    }
 else inv_n |= HI_BIT;

 return inv_n / MAXHLT1;
}
```